A study of the design and documentation skills of industry-ready CS students*

Mrityunjay Kumar
IIIT Hyderabad
Hyderabad, Telangana, India
mrityunjay.k@research.iiit.ac.in

Venkatesh Choppella
IIIT Hyderabad
Hyderabad, Telangana, India
venkatesh.choppella@iiit.ac.in

ABSTRACT

An engineer in a product company is expected to design a good solution to a computing problem (Design skill) and articulate the solution well (Expression skill). We expect an industry-ready student (final year student or a fresh campus hire) as well to demonstrate both these skills when working on simple problems assigned to them.

This paper reports on the results when we tested a cohort of participants (N=16) for these two skills. We created two participant groups from two different tiers of college, one from a Tier 1 college (who were taking an advanced elective course), and another from Tier 2 colleges (who had been hired for internship in a SaaS product company). We gave them a simple design problem and evaluated the quality of their design and expression. Design quality was evaluated along three design principles of Abstraction, Decomposition, and Precision (adapted from the Software Engineering Book of Knowledge). Expression quality was evaluated using criteria we developed for our study that is based on the diversity and density of the expressions used in the articulation.

We found the students lacking in design and expression skills. Specifically, a) they struggled with abstraction as a design principle, b) they did not use enough modes of expressions to articulate their design, and c) they did not use enough formal notations (UML, equations, relations, etc.). We also found significant difference in the performance between the two participant groups.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → **Software design engineering**.

KEYWORDS

software design skills, problem-solving skills, industry-ready students, engineering productivity

*CS undergraduate students who are in fourth year or beyond are referred to as industry-ready in this paper.

COMPUTE '22, November 09–11, 2022, Jaipur, India

ACM ISBN 978-1-4503-9775-9/22/11…$15.00
https://doi.org/10.1145/3561833.3561842

ACM Reference Format:
Mrityunjay Kumar and Venkatesh Choppella. 2022. A study of the design and documentation skills of industry-ready CS students*. In *COMPUTE 2022 (COMPUTE 2022), November 9–11, 2022, Jaipur, India.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3561833.3561842

1 INTRODUCTION

The Indian product startup ecosystem continues to grow. In 2021 alone, there were more than 2200 startups founded, and tech startups raised more than 24B USD [16], including many SaaS startups [19]. These companies require a strong pool of software engineers and industry-ready students. However, the quality of students graduating from engineering colleges is poor; less than 4% of the engineers have enough skills for these startups and product companies [1, 5].

Design is a key step in an engineering process [21], and the ability to design is a key skill for a software engineer. This translates into three competencies that companies seek: 1) ability to design solutions to problems, 2) ability to articulate the design, and 3) collaborate with the team on the design and make it fit for the company. [3, 10].

Given the need for skilled engineers by startups and product companies, and given that design ability is a key skill, we want to understand the level of design skills in industry-ready students (CS undergraduate students who are in fourth year or beyond) of the engineering colleges.

SWEBOK [7] lists seven software design principles: 1) Abstraction, 2) Coupling and Cohesion, 3) Decomposition and Modularization, 4) Encapsulation and Information Hiding, 5) Separation of interface and implementation, 6) Sufficiency, Completeness, and Primitiveness, and 7) Separation of concerns. To restrict our scope, we selected three principles from this list to evaluate the students on. **Decomposition** reflects the ability to break down a large problem into several small ones that can be independently solved and composed into a solution for the large problem [22]. **Abstraction** [14] is the ability to generalize and focus on essential information, either through parameterization or specification. **Precision** (Sufficiency or Completeness) is the ability to handle all stated requirements correctly with the proposed design.

For our experiment, we define Design Skill as follows:

Design Skill: *Ability to apply key design principles of Abstraction, Decomposition and Precision in a solution design*'.

This paper presents our findings from an experiment we ran with two participant groups to understand their Design Skill.

We evaluated the participant submissions on two dimensions: a) Design Quality, which evaluated the design against the three design principles, and b) Expression Quality, which evaluated the design

against the usage of different ways of expressing their design - text, pictures, notations, etc.

Research Questions. We proceeded to answer three research questions about the design produced by the participants:

> *RQ1: Do industry-ready students produce good quality design?*
> *RQ2: Do industry-ready students articulate their design well?*
> *RQ3: Is there a difference in performance between Tier 1 and Tier 2 college students?*

The rest of the paper is organized as follows: Section 2 presents related work on design skills. Section 3 describes the design of the experiment. Section 4 lays out the evaluation criteria and other details about the evaluation. Section 5 discusses the findings based on the analysis of the results. Finally, Section 6 presents our conclusion and plan for future work.

2 RELATED WORK

The software industry is a knowledge-intensive industry. As Robillard [18] says, '*Software development is the processing of knowledge in a very focused way*'. This processing applies the knowledge to solve the computational problem at hand, and such skills are required for industry-ready students to do well in their job. A precise definition of what is software design may be elusive [23], but their importance is well-established [21]. Hewner and Guzdial [10] showed, albeit in a limited game company setting, that "*Being able to solve algorithmically challenging problems*" and "*Ability to build a good object design for a large system and understand the implications*" are held as very important skills for the industry. Radermacher et al [17] showed that '*Problem solving*' is in top 5 knowledge competencies identified in recently graduated students. Hartman et al [9] mention '*The two most important skills which a student embarking on his career can have are communication skills and problem solving skills.*'

Of the seven design principles laid out in SWEBOK [7] we chose Abstraction, Decomposition, and Sufficiency as key skills to evaluate for these industry-ready students.

Abstraction. Kramer [11] strongly argues that Abstraction is key to computing. Liskov [13, 14] and others [24] have suggested many approaches to achieve abstraction in the design (abstraction by parameterization and by specification are two key ones) and we expect the students to demonstrate this ability.

Decomposition. Hoek and Lopez [22] claim that "..Without [Modularization], large software systems simply could not be realized". Various decomposition and modeling techniques (functional, subsystem-based, abstraction, etc.) have been in use and can be applied when one designs a system [8, 15], and we expect the industry-ready students to evidence modularity and decomposition in their design.

Precision. Sufficiency and completeness of the design ensures there is less rework required after release. Precision can be usually improved by leveraging formal visual languages like UML [6], or algebraic-logical languages like Z notation [20]. We expect the industry-ready students to use some of the tools to achieve precision without sacrificing abstraction.

3 EXPERIMENT SETUP

3.1 Study Design

We asked students to devise a design for their solution for a given problem and then describe their design in 2-3 pages. The problem was given as a take-home assignment to two groups of students with 7 days to complete it, so there was enough time given for the students to do a good job. We wanted to select a simple problem that allows the elements of abstraction, decomposition, and precision to a level we expect an industry-ready student to possess. We selected a board game (Tic-tac-toe) that is usually used as a non-trivial programming problem for students to tackle as a beginner programmer. While it is hard to show any selected problem is the best choice, Tic-tac-toe solution design demonstrates abstraction, decomposition, and precision well (see the solution outline bit.ly/compute22sol) so we believe it is a good choice of a design problem for the students.

Problem summary: We asked them to consider the 2-person game tic-tac-toe in which a human plays the game with a computer. They were then asked to think about the design of a software solution for this and then describe the design in writing, with enough precision that a first year CS student can code against. We also asked them not to write code or pseudo-code and stay at the design level. Complete problem text that was given to them: bit.ly/compute22prob

3.2 Participants

We selected two types of participants for this experiment. **Group A** consisted of third- and fourth-year students of a Tier 1 college who were enrolled for a "Topics in Software Foundations" course (Computer Science Elective). This design problem was given as part of their graded assignments. **Group B** consisted of students from good Tier 2 college who had gone through two rounds of interview for internship at a local SaaS (unicorn) company and were selected for internship.

Incentive for good work: To ensure that students and interns do their best on these problems, this was given as part of the graded assignment to Group A participants. The first author delivered a Design Masterclass for Group B participants as part of their training and this design problem was given as pre-requisite for attending this workshop.

4 EVALUATION SETUP

As mentioned in the Introduction, we evaluated the submissions on two dimensions: Design Quality and Expression Quality. Participants submitted their work as PDF documents that captured their design, and we used expert evaluation (by author) of the submissions to generate the score for analysis.

4.1 Design Quality Evaluation

The submissions were evaluated on the three selected design principles:

(1) **Abstraction**
(2) **Decomposition**
(3) **Precision**

We then created a rubric to evaluate the evidence of each of these design principles in the submission, using the solution outline created earlier (bit.ly/compute22sol).

4.1.1 Decomposition rubric. Rubric: We evaluate based on how many of the components (as identified in solution outline or similar) are identified as part of decomposition and used the score of 0 (No components identified), 1 (one or two components identified), and 2 (three or more components identified).

4.1.2 Abstraction rubric. According to Liskov [13, 14], abstraction can be achieved in two ways: a) *Abstraction by Parameterization* - The system should demonstrate that general-purpose functions, procedures, and services have been designed and used, and b) *Abstraction by Specification* - The system should demonstrate that services and functions are defined such that interface definitions are enough to use them.

Abstraction is a 'good' way to decompose - so if decomposition is not done well (say only one component is identified), abstraction will not be in evidence. To ensure we do not penalize twice, we only look for abstraction in the defined components and give full marks if it is done well (even when the remaining components are missing). We also want to ensure that the students do not resort to low level details like code or pseudo-code and so we penalize it.

Rubric: We evaluate based on evidence of abstraction and use scores 0 (No abstraction evidenced), 1 (1-2 methods identified for the components and/or some interface definitions but mostly incomplete), and 2 (Complete interface definitions for components, the methods on the components are sufficiently specified or described).

4.1.3 Precision rubric. We listed down key scenarios and evaluated the submissions against these: a) The system declares a result correctly in all cases (win or draw), b) The system has a move for every valid move from the human player, c) The system's move is always the most optimal in that situation, d) The board state is always available, and e) The board state is available correctly after every move.

Result of an evaluation of a scenario can be Pass (the description can correctly handle the scenario), Inconclusive (not enough information to say pass or not) or Fail (incorrect handling of the scenario by the description).

Rubric: We evaluate based on how many scenarios Pass, and use scores of 0 (None of the scenarios pass), 1 (50% of the scenarios pass), and 2 (All scenarios pass).

4.1.4 Scoring. Design quality was scored using the rubrics for Abstraction (section 4.1.2, Decomposition (section 4.1.1) and Precision (section 4.1.3) To reflect the relative prioritization across these three parameters, we assigned different weights to the parameters: Decomposition, Abstraction, and Precision were Fibonacci-weighted (2, 3, 5, respectively) to reflect the fact that abstraction is more important than just decomposition, and precision is more important than both. Average scores within each parameter were normalized to 100% to compare them across groups and across parameters. Table 2 presents the scores and averages for the participants and the groups.

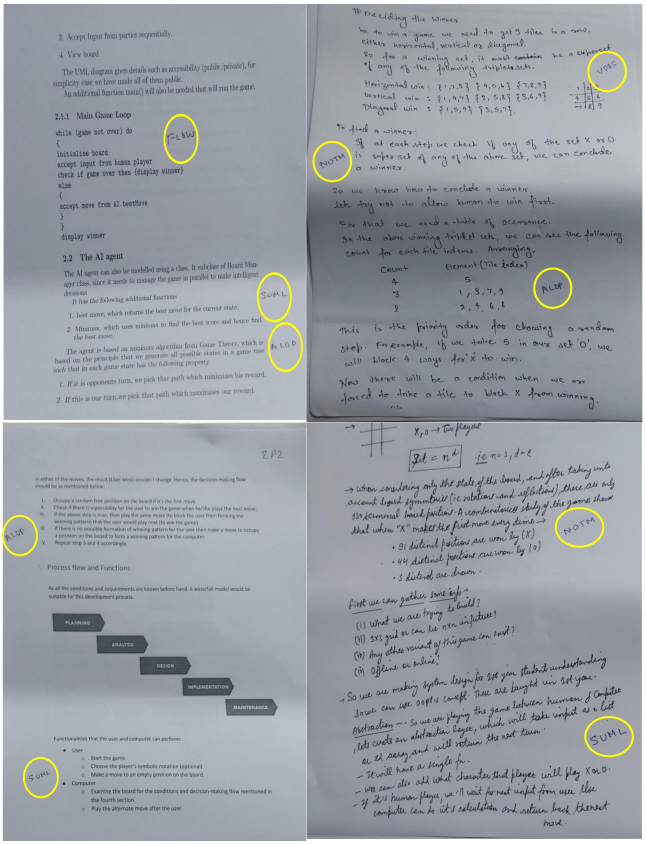

Figure 1: Encoded submission samples (codes are yellow-circled)

We also calculated the pair-wise correlation among the Decomposition, Abstraction and Precision scores of the Group A participants (Table 1). This was not done for Group B participants, since Precision scores for all samples were 0.

4.2 Expression Quality Evaluation

An expression is a linguistic or domain-specific means to describe something (in this case, design) - like text, diagrams, flow charts, UML, etc. This is also called mode, and social scientists and linguists refer to it as multimodal text (we use 'expression' in this paper).

We encoded the submission to identify the expression types they used. The submission was broken into chunks (a text paragraph or a picture) and each chunk was encoded using one of the codes defined (section 4.2.1). See Figure 1 for some samples of encoding of the submissions. We generated two metrics from the encoding.

- **Diversity** measures how many distinct expression types were used
- **Density** counts total number of expressions used irrespective of type.

These two metrics were reported for each student (Table 2). Technically, this measures quantity and not quality. However, it has been shown that using multimodal texts (expressions) improve the quality of the text [2, 4]. For this paper, we have used these two

metrics as proxy to quality of their expression. We expect that every participant uses each of these expression types at least once, which translates to a Diversity average of 7 (everyone uses all types).

4.2.1 Expression types. This is the list of the expressions we looked for. The 4-letter code is the representation during encoding.

- Visual Description, Diagrams (VDES)
- Procedures and Interface definitions (PROC)
- Pre-defined abstractions like algorithms and design patterns (ALDP)
- Flow Chart or steps description (FLOW)
- Static modeling diagrams - UML or similar - for Class, Object, Component depiction etc. (SUML)
- Behavioral or Dynamic modeling diagrams - UML or similar - for Activity, Sequence, States etc. (BUML)
- Modeling and Specifications through notations like Z-notation, Sets/relations, formalized English, etc. (NOTM)

5 FINDINGS AND DISCUSSION

(Abbreviations used in this section. ***DC**: Decomposition, **AB**: Abstraction, **PR**: Precision, **CWS**: Composite Weighted Score, **DV**: Expression Diversity, **DN**: Expression Density, **M**: Mean, **SD**: Standard Deviation, **CoV**: Coefficient of Variance*)

We set out to understand the Design Quality (RQ1) and Expression Quality (RQ2) of the participants, and whether there is a difference in performance between the two groups (RQ3).

Mean Composite Weighted Score for the two groups (together) is 3.06 (out of 10), which is quite low (barely passing marks!). This suggests that **Design Quality for this cohort is poor**. We find that students used a small number of expression types (DV Mean: 2.69) and number of expressions (DN average: 4.69) when articulating their design. We expected all types to be used and many more expressions to be used on an average, so **Expression Quality is poor as well**. We expected the groups to have similar performance. However that is not the case; **there is a difference in performance between Group A and B**. We find that a) Group B has significantly worse performance than Group A (CWS average: 13.1% vs. 48.1%), b) Precision was hard to achieve for everyone (PR average: 43.8%) but worse for Group B (0%), c) Abstraction had moderately positive correlation to Decomposition and Precision for Group A (0.64), and d) we find that there was very little use of behavioral/dynamic modeling specification (BUML M- 0.38, CoV: 2.15) or formal notations for modeling/specification (NOTM M: 0.5, CoV: 1.79) (Table 4).

5.1 Group B has significantly worse performance than Group A

As seen in Table 3, Mean Composite Weighted Score for the entire set is 3.06 (out of 10), which is quite low. Group A (Tier 1 college students) has this at 4.81 and Group B (Tier 2/3 college interns at a company) at 1.31. This suggests that Group B's performance has been significantly poorer compared to Group A (Table 2). Precision similarly shows significant difference in performance between A and B (0.88 vs. 0). In addition, Coefficient of Variance (CoV) for Group A lie between 0.53 and 0.95, while those for Group B are higher (1.18 to 1.38), which suggests more variability among the Group B students.

Table 1: Pair-wise correlation for Group A scores

	Decomposition	Abstraction	Precision
Decomposition	1	0.64	0.44
Abstraction	0.64	1	0.64
Precision	0.44	0.64	1

5.2 Precision (Completeness, Sufficiency) was hard to achieve for everyone

Very few students (<22%) were able to get the design to cater to all the scenarios; Group B didn't get any scenarios to pass. Group A fared much better (Mean: 0.88). However, even for Group A, Coefficient of Variance (CoV) of 0.95 is high which suggests large variation among students. Lack of precision can be either because of lack of design quality or it can be due to lack of expression quality. A focus on a consistent structure of the documentation and being able to abstract well enough could help improve this score.

5.3 There wasn't enough expression diversity or density

Using more expression types (diversity) like visual descriptions, flow charts, notations, Static UML, etc. and using more expressions in general (density) improves the quality of the description [2, 4]. However, the documentation produced by the students used very few expression types (DV M:2.69,CoV:0.52), and very small number of expressions (DN M:4.69,CoV:0.58), resulting in a text-heavy and less clear documentation. We expected almost all the expression types to be used by all students.

5.4 There was a lack of formal notation usage for capturing dynamics of the design

This was a surprising finding across the entire set of students. Dynamics of the system (how the system moves from one state to another) is a key aspect of design and needs to be captured formally to improve precision, Li et al. [12] analyzed two large open-source software systems and showed that more than 80% of defects are semantic (functionality related) in nature. However, less than 20% of the expression usage included Behavioral UML or any set theoretic notations. Behavioral modeling aspect of UML was used quite less (BUML M: 0.38) and variation was large amongst participants (BUML CoF: 2.15). Same was the case for any notations/specifications (NOTM M: 0.5 CoF: 1.79). Not using these also caused lack of precision for many students.

5.5 Abstraction may be a key skill gap to fill

A mean of 1 (out of 2) for Group A on Abstraction indicates that participants didn't do well. Correlation data (Table 1) shows abstraction has moderately positive correlation with Decomposition (0.64) as well as Precision (0.64) for Group A. This correlation was not expected and is worth investigating. If this correlation can point to causation, it is significant because it suggests that picking abstraction skills may help improving decomposition and modularization skills and may also make the design more precise.

Table 2: Design evaluation scores (N=16)

Name	Grp	DC	AB	PR	CWS	DV	DN
CP1	A	1	1	1	5	3	7
CP2	A	0	0	0	0	1	1
CP3	A	2	1	0	3.5	2	3
CP4	A	2	2	2	10	3	6
CP5	A	1	1	2	7.5	3	3
CP6	A	1	1	1	5	5	9
CP7	A	0	1	0	1.5	4	7
CP8	A	2	1	1	6	4	7
ZP1	B	0	0	0	0	3	5
ZP2	B	1	0	0	1	4	5
ZP3	B	2	1	0	3.5	1	2
ZP4	B	2	1	0	3.5	3	6
ZP5	B	0	0	0	0	1	1
ZP6	B	0	0	0	0	1	1
ZP7	B	1	1	0	2.5	3	5
ZP8	B	0	0	0	0	4	8
Avg(all)		46.9%	34.4%	21.9%	30.6%	2.8	4.8
Avg(A)	A	56.3%	50.0%	43.8%	48.1%	3.1	5.4
Avg(B)	B	37.5%	18.8%	0.0%	13.1%	2.5	4.1

***DC**: Decomposition, **AB**: Abstraction, **PR**: Precision, **CWS**: Composite Weighted Score, **DV**: Expression Diversity, **DN**: Expression Density*

Table 3: Design Evaluation Summary Statistics (N=16)

	All			Grp A			Grp B		
Name	M	SD	CoV	M	SD	CoV	M	SD	CoV
DC	0.94	0.85	0.91	1.13	0.83	0.74	0.75	0.89	1.18
AB	0.69	0.60	0.88	1.00	0.53	0.53	0.38	0.52	1.38
PR	0.44	0.73	1.66	0.88	0.83	0.95	0.00	0.00	-
CWS	3.06	3.04	0.99	4.81	3.20	0.66	1.31	1.60	1.22
DV	2.69	1.40	0.52	3.00	1.51	0.50	2.38	1.30	0.55
DN	4.69	2.73	0.58	5.25	2.96	0.56	4.13	2.53	0.61

***DC**: Decomposition, **AB**: Abstraction, **PR**: Precision, **CWS**: Composite Weighted Score, **DV**: Expression Diversity, **DN**: Expression Density*
***M**: Mean, **SD**: Standard Deviation, **CoV**: Coefficient of Variance*

Table 4: Expression Types Summary Statistics (N = 16)

Code	Description	M	SD	CoV
VDES	Visual Description (Diagrams)	1.06	1.24	1.16
PROC	Procedures, Interfaces	0.25	0.58	2.31
ALDP	Pre-defined abstractions (algorithms, design patterns)	0.38	0.50	1.33
FLOW	Flow Chart or description	0.81	1.05	1.29
SUML	Static modeling diagrams - UML or similar (Class, Object, Component)	1.31	1.45	1.10
BUML	Behavioral/Dynamic modeling diagrams - UML or similar (Activity, Sequence, Statechart)	0.38	0.81	2.15
NOTM	Modeling/Specification through notations (Z-notation, Sets/relations, English, etc.)	0.50	0.89	1.79

***M**: Mean, **SD**: Standard Deviation, **CoV**: Coefficient of Variance*

6 CONCLUSION AND LIMITATIONS

We evaluated 16 students' attempts to design a simple system and describe the design. Overall, the students demonstrated poor design and expression skills. Data suggests that most students found it hard to achieve precision in their design. Of those who did well in being precise, most of them also did well in abstracting their design. Structure of these design documents were inconsistent, and this contributed to lack of precision as well, in addition to lack of usage of formal specification mechanisms like UML. This suggests lack of basic skills in technical documentation and articulation skills. These students came from two different college tiers and the difference between their performance in this experiment is significant.

From teaching and learning perspective, there are a few areas where there should be focused intervention: a) Use abstraction well to achieve a good design, b) Use formal notations to achieve precision in the design, and c) Use defined structure and approach to produce good design documentation.

This also suggests why we have trouble producing employable engineers. If these basic skills are missing, we can't expect these industry-ready students to do well in industry. The contribution of this paper is in defining a way to evaluate these attributes and point towards areas to focus teaching and learning efforts in.

Limitations. These limitations of this study should be kept in mind when interpreting and using the results: a) This study was done on a small sample size (N=16), b) The difference between Group A and Group B performance can have many other factors confounding the results which were not controlled for (incentives may not be strong for Group B to perform better, for example, or the selection of students from these colleges.), c) The first author acted as the sole expert for analysis and this may have introduced a researcher's bias in the coding and scoring, and d) The rubric for expression quality evaluation is quite subjective.

Future work. We plan to run the experiment with more cohorts across different college categories and refine the encoding and rubric criteria. We also plan to identify curriculum interventions to enable students to acquire these skills effectively and efficiently.

Applicability. Even with this small sample size, a few key skill gaps in students are apparent and instructors can plan to focus on these: a) students struggle in abstracting well, b) students do not use enough diversity and density of expressions to articulate their design well, and c) students do not use enough formal notations of any kind. Teachers can help students pick these skills through their course assignments. In a subsequent paper, we plan to report on approaches to address these gaps through pedagogical interventions.

REFERENCES

[1] Varun Aggarwal, Siddharth Nithyanand, and Malvika Sharma. 2020. National Employability Report for India - 2019. https://www.shl.com/en-in/resources/by-type/whitepapers-and-reports/national-employability-report-engineers-2019/

[2] Lasisi Ajayi. 2009. English as a second language learners' exploration of multimodal texts in a junior high school. *Journal of Adolescent & Adult Literacy* 52, 7 (2009), 585–595.

[3] Andrew Begel and Beth Simon. 2008. Novice Software Developers, All over Again. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1404520.1404522

[4] Jeff Bezemer and Gunther Kress. 2008. Writing in multimodal texts: A social semiotic account of designs for learning. *Written communication* 25, 2 (2008), 166–195.

[5] Andreas Blom and Hiroshi Saeki. 2011. Employability and skill set of newly graduated engineers in India. *World Bank Policy Research Working Paper* 5640 (2011).

[6] Grady Booch. 2005. *The unified modeling language user guide.* Pearson Education India.

[7] P Borque and R Fairley. 2014. Guide to the Software Engineering Body of Knowledge Version 3.0. *IEEE Computer Society Staff* (2014).

[8] David Faitelson and Shmuel Tyszberowicz. 2017. Improving Design Decomposition (Extended Version). *Form. Asp. Comput.* 29, 4 (jul 2017), 601–627. https://doi.org/10.1007/s00165-017-0428-0

[9] Janet Hartman and Curt M. White. 1990. "Real World" Skills vs. "School Taught" Skills for the Undergraduate Computer Major. *SIGCSE Bull.* 22, 1 (feb 1990), 216–218. https://doi.org/10.1145/319059.323455

[10] Michael Hewner and Mark Guzdial. 2010. What Game Developers Look for in a New Graduate: Interviews and Surveys at One Game Company. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 275–279. https://doi.org/10.1145/1734263.1734359

[11] Jeff Kramer. 2007. Is Abstraction the Key to Computing? *Commun. ACM* 50, 4 (apr 2007), 36–42. https://doi.org/10.1145/1232743.1232745

[12] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on*

Architectural and system support for improving software dependability. 25–33.

[13] Barbara Liskov and John Guttag. 2000. *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education.

[14] Barbara Liskov, John Guttag, et al. 1986. *Abstraction and specification in program development*. Vol. 180. MIT press Cambridge.

[15] Ali Najafi, Nan Niu, and Farzaneh Najafi. 2011. Multi-Level Decomposition Approach for Problem Solving and Design in Software Engineering. In *Proceedings of the 49th Annual Southeast Regional Conference* (Kennesaw, Georgia) *(ACM-SE '11)*. Association for Computing Machinery, New York, NY, USA, 249–254. https://doi.org/10.1145/2016039.2016104

[16] NASSCOM. 2021. NASSCOM Tech start-up report 2021 – year of the Titans. https://nasscom.in/knowledge-center/publications/nasscom-tech-start-report-2021-year-titans

[17] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap between Graduating Students and Industry Expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/2591062.2591159

[18] Pierre N. Robillard. 1999. The Role of Knowledge in Software Development. *Commun. ACM* 42, 1 (jan 1999), 87–92. https://doi.org/10.1145/291469.291476

[19] SaaSBooMi. 2021. India's SaaS Landscape Report 2020. https://saasboomi.com/saas-landscape-report/

[20] J Michael Spivey. 1988. *Understanding Z: a specification language and its formal semantics*. Vol. 3. Cambridge University Press.

[21] Richard N Taylor and Andre Van der Hoek. 2007. Software design and architecture the once and future focus of software engineering. In *Future of Software Engineering (FOSE'07)*. IEEE, 226–243.

[22] André van der Hoek and Nicolas Lopez. 2011. A Design Perspective on Modularity. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development* (Porto de Galinhas, Brazil) *(AOSD '11)*. Association for Computing Machinery, New York, NY, USA, 265–280. https://doi.org/10.1145/1960275.1960307

[23] Giovanni Viviani and Gail C Murphy. 2022. What really is software design?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 301–308.

[24] Yingxu Wang. 2008. A Hierarchical Abstraction Model for Software Engineering. In *Proceedings of the 2nd International Workshop on The Role of Abstraction in Software Engineering* (Leipzig, Germany) *(ROA '08)*. Association for Computing Machinery, New York, NY, USA, 43–48. https://doi.org/10.1145/1370164.1370174